ELSEVIER

Theoretical Computer Science 281 (2002) 207–217

Theoretical Computer Science

www.elsevier.com/locate/tcs

Tilings as a programming exercise

Guy Cousineau

PPS Laboratory, Université Denis Diderot, 175 rue du Chevalret, 75013 Paris, France

Abstract

We investigate the problem of producing symmetric tilings by programs in a uniform way. By this, we mean that the construction of a tiling should be parameterized by the geometric model in which the tiling is defined (Euclidean, Hyperbolic, etc.), its formal symmetry group, the interpretation of this symmetry group in the geometric model and a specific tile that fits this interpretation.This parameterization can be obtained at a basic level using higher order functions as described in Cousineau and Mauny (The Functional Approach to Programming, Cambridge University Press, Cambridge, 1997) but this fails to reflect in the program the mathematical structure of tilings in an adequate way. We therefore explore the use of more abstract constructs such as modules and show that they allow to structure the program in a satisfying way. The Objective Caml programming language (Objective Caml, O'Reilly, 2000) allows us to place this study in a convenient setting. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Functional Programming; Modularity; Parametricity; Tilings

1. Introduction

The use of computers in Mathematics is now widespread both in research and in education. Numerous applications such as Mathematica [9], or Maple [7] are very popular. Many more specialized applications such as GAP [5] are developed for research purposes and gather important communities of developers and users. These systems include a programming language in which each user can develop specific programs. This programming language is supposed to give access to mathematical theories in a way that is convenient from a programming point of view and this involves questions of abstraction, modularity, parametricity that are central also in the design of programming languages. Therefore, the design of such systems involves problems that are clearly relevant to Computer Science. When this aspect is ignored or underestimated, this can restrain the use of computer mathematics. This problem is indeed addressed by some

E-mail address: guy.cousineau@pps.jussieu.fr (G. Cousineau).

PII: S0304-3975(02)00013-0

research groups in Computer Science (e.g. project FOC at University of Paris VI) but this has not led yet to widespread systems since the problems involved are indeed deep and difficult.

In this paper, our ambition is much more limited. We only intend to show how an application involving rather simple concepts from geometry and group theory can be programmed with a satisfying generality using a modern programming language with a well-designed module facility. We hope to convince the reader, that such a facility is indeed necessary to reflect in programs the right level of abstraction for this kind of application. We address the problem of producing tilings [6] with the help of a computer. The tilings we consider are those with a non-trivial computable symmetry group acting transitively on the tiles. This includes periodic tilings of the Euclidean plane but also other kinds of tilings, for instance hyperbolic ones such as the Escher-like tilings in Fig. 4.

The mathematical concepts involved in the design of such tilings are

- Symmetry groups and their formal presentations
- Geometric transformations that can interpret such groups in a given geometry
- Geometric objects that can be submitted to such transformations such as points and lines or curves.

Our general approach is that we produce a tiling by computing a finite part of its symmetry group, interpret the obtained elements of this symmetry group as transformations in a given geometry and finally apply these transformations to a basic tile or pattern to produce the tiling.

2. Design problems

The main problem we have to face is parametricity. The objects we deal with are highly structured and refer to mathematical concepts that enable us to describe a tiling in an adequate way. Therefore, an implementation should reflect this structure and our program should be parameterized by the involved mathematical objects such as a symmetry group or a geometry.

Moreover, a tiling has many concrete aspects that are important for its final aspect such as colors, decorations, contour lines but that should not interfere with the overall generating process. This causes some difficulties. For instance, for a tiling to be colored in an adequate way, the transformations that are used to produce the different copies of a tile should be paired with permutations that act on the colors. There are also variations that can be useful occasionally. For instance, it might be useful for pedagogical purposes to mark tiles by labels that indicate by which transformation a given tile has been produced. In that case, transformations should be paired with labels describing them as in Fig. 1. This sort of variation should be easy to make and should not imply modifications of the generating process. We shall first examine what this implies for the representation of tiles.

To draw tiles, we must first assume that we can use a graphical library. We shall call pictures the graphical objects produced by that library. However, it would not be adequate to represent directly tiles by pictures because it is very unlikely that

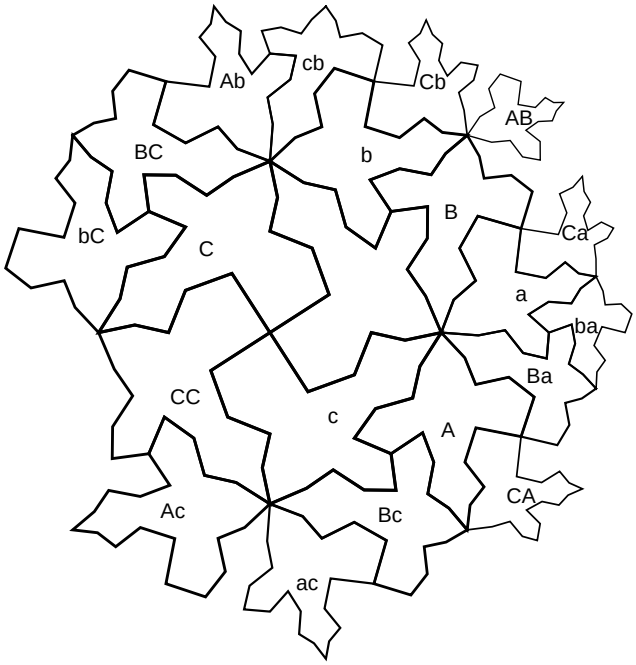

Fig. 1. Escher's Circle Limit III with labels.

the graphical library provides all the transformations that we might have to apply to tiles. For instance, when tiles are tagged and a tile is rotated, its tag should not be rotated. So a tile should have more structure than a picture. On the other hand, a tile should be something that can produce a picture when extra information is passed to it.

A plausible solution is to consider that a tile is a function which takes that extra information and produces a picture. We shall call transformation such extra information which concerns position and shape (geometrical transformation), color (color permutation) or any other characteristics. We can summarize this point of view by a type statement

```
tile: transformation -> picture
```

However, as we already mentioned, transformation cannot be a fixed type because various kinds of transformations can be used for various tilings. So the type of tiles should depend on a specific transformation type which is part of the tile specification. This is possible only if our programming languages allows us to specify such dependencies. We can also notice that a tile is in general a rather complex geometrical object. So, if we define it as a function, this function will typically make use of some data structure related to it. It might be tempting to consider tiles as objects in the sense of object oriented languages of which the above function would be a method. But then

the aforementioned type dependency should be expressed by a class dependency, a notion that does not exist in object-oriented languages.

Now, let us take a look at tilings. To produce a tiling, we need both a tile (in the above sense) and a process that can generate transformations to be passed to the tile. We shall call generator such a process. Here again, the type of a generator should depend on the type of transformations it generates and moreover, when a generator is used together with a tile to produce a tiling, it is mandatory that both depend on the same transformation type. This kind of constraints should also be expressible in the programming language.

Furthermore, generators will have to be constructed in a uniform way from a group presentation and an interpretation of that group in a given geometry and the transformation type it will use comes from the geometry. So we need to be able to define some functions that take objects with rather complex type dependancies and produce objects with similar characteristics.

We shall solve these problems by using the module system of the OCAML language which comes originally from the module system of the SML language with some additions. In this system, modules are complex entities which incorporate types and values. One can also define functors which build new modules from arguments that are also modules and express constraints between arguments of functors.

3. Basic modules and functors for tilings

Here is a module type definition for tiles:

```
module type TILE =
sig
 type transformation
 val visualize : transformation -> picture
end
```

This definition states that a tile (a module of type TILE also called a module with signature TILE) includes a type transformation on which the function visualize depends. A tile will contain other fields but these two will be the only visible ones from the signature TILE.

A generator will have the following type:

```
module type GENERATOR =
sig
 type element
 val generate : int -> element list
end
```

A generator includes a type element and a function generate that produces a list of elements given an integer. This integer will typically allow to specify how much of a symmetry group one wants to generate.

A tiling will have the following type:

```
module type TILING =
sig
 val draw : int -> picture
end
```

Now, the following functor will produce a tiling from a generator and a tile. It includes a constraint saying that the elements generated by the generator should be transformations accepted by the tile.

```
module Make_tiling
 (Gen: GENERATOR)
 (Tile: TILE with type transformation = Gen.element)
=
struct
 let draw n = group_pictures (map Tile.visualize (Gen.generate n))
end
```

4. Group representation

One of the problems we have to face when generating symmetry groups of tilings is to do it in a non-redundant way. Fortunately, common symmetry groups have presentations with decidable word problems [4] and moreover, they have presentations with canonical rewriting systems as shown by Lechenadec in [8]. Therefore, the generation of normal forms are quite straightforward: the presentation of the group can be given as a function that takes a generator and a list of generators and adds the new generator to the list provided that this does not introduce a redex for the associated rewriting system. So, the module type for canonical groups is:

```
module type CANONICAL_GROUP =
sig
 type element
 val generators : element list
 val compose : element -> element list -> element list
end
```

For instance, the canonical presentation for Escher's Circle Limit symmetry group is an hyperbolic group with 3 rotations of order 3, 3 and 4. If we denote A, B, C the 3 rotations and a, b, c their inverses, the corresponding rewriting system is the following (omitting trivial rules such as $Aa \to Id$).

$AA \to a$	$AC \to b$	$BA \to c$	$BB \to b$
$CB \to a$	$CCC \to c$	$aB \to Cb$	$aC \to Ab$
$aa \to A$	$ab \to C$	$bA \to Bc$	$bCC \to Ac$
$bb \to B$	$bc \to A$	$cA \to Ba$	$cB \to CCa$
$ca \to B$	$cc \to CC$		

The corresponding module is defined below with the rotations written TA, TB and TC and their inverses Ta, Tb and Tc.

```
module Group_hyp_3_3_4  =
struct
 type element = TA | TB | TC | Ta | Tb | Tc
 let generators = [TA;TB;TC;Ta;Tb;Tc]
let compose = function
  TA -> (function []->[TA] | ((TB::_)|(Tb::_)|(Tc::_) as tl')->TA::tl')
| TB -> (function []->[TB] | ((TC::_)|(Ta::_)|(Tc::_) as tl')->TB::tl')
| TC -> (function []->[TC]
          | ((TA::_)|(Ta::_)|(Tb::_)|((TC::(TA|TB|Ta|Tb|Tc)::_)) as tl')
          -> TC::tl')
| Ta -> (function []->[Ta] | ((Tc::_) as tl')->Ta::tl')
| Tb -> (function []->[Tb]
           | ((Ta::_)|((TC::(TA|TB|Ta|Tb|Tc):: _)) as tl')->Tb::tl')
| Tc -> (function []->[Tc] | ((Tb::_) as tl')->Tc::tl')
end
```

To each canonical group, we associate a generator using the following functor in which we do not detail the function power which produces an exponential from a product and its neutral element.

```
module Make_canonical_generator (G:CANONICAL_GROUP) =
struct
 type element = G.element list;;
 let power f x l = ...
let generate n = power G.compose [] G.generators n
end
```

We also need a representation for concrete groups which are defined by

```
module type GROUP =
sig
 type element
 val compose : element -> element -> element
 val identity : element
end
```

In fact, we only use the monoidal structure of groups, so the inverse function is not included.

5. Mappings

Next, we have to interpret elements of formal symmetry groups as geometric transformations using a morphism from formal groups to concrete transformation groups. A morphism is defined by giving a mapping from the generators of the formal group to geometric transformations and this mapping can then be automatically extended to a morphism. Here again, modules and functors are a great help in implementing these notions at a suitably abstract level. A mapping is just a function from a source to a destination, depending on the source and destination type.

```
module type MAPPING =
sig
 type source
 type dest
 val map : source -> dest
end
```

A useful construct is the pairing of mappings which is implemented by the functor:

```
module MakePair (Map1: MAPPING)
                (Map2: MAPPING with type source = Map1.source)
 =
struct
  type source = Map1.source
  type dest = Map1.dest * Map2.dest
  let map s = Map1.map s , Map2.map s
end
```

A morphism from canonical groups to concrete groups is computed from a mapping of its generators by the following functor:

```
module Make_morphism (G1:CANONICAL_GROUP)
                     (T:MAPPING with type source = G1.element)
                     (G2:GROUP with type element = T.dest)
=
struct
  type source = G1.element list;;
  type dest = G2.element;;
  let rec map = function [] -> G2.identity
                   | a::l  -> G2.compose (T.map a) (map l);;
end
```

What we obtain is a new mapping that involves lists of generators instead of simple generators. Note how the constraints on the arguments of the functors allow to specify in a simple way the type dependencies.

It is now rather easy to take into account other aspects of a tiling. For instance, we can map the elements of a canonical group presentation to permutations using this functor:

```
module Make_color_morphism (G1:CANONICAL_GROUP)
                (C:MAPPING with type source = G1.element
                           and type dest = Permutation.permutation)
=
struct
  type source = G1.element list;;
  type dest =  Permutation.permutation;;
  let rec map = function [] -> Permutation.identity
                   | a::l  -> Permutation.compose (C.map a) (map l);;
end
```

and pair the two obtained mappings using the functor MakePair.

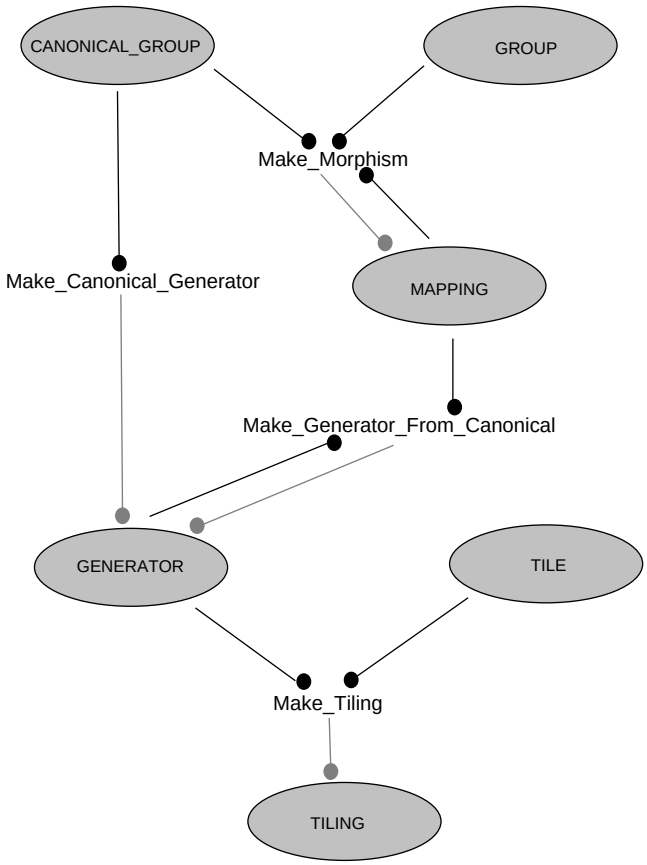

Fig. 2. Modules and functors.

The next step is to use such a morphism to transform a canonical generator into a generator for a concrete group. We just need a functor that takes a generator and a mapping and produces a new generator:

```
module Make_generator_from_canonical
 (Gen: GENERATOR)
 (M: MAPPING with type source = Gen.element)
=
struct
 type element = M.dest;;
 let generate n = List.map M.map (Gen.generate n);;
end
```

All the ingredients to produce tilings are now defined and are shown in Fig. 2.

6. Geometries

To structure a bit more the construction of tilings, we introduce a last module type which corresponds to geometries. A geometry depends on a type point and a type transformation depending on point. It essentially introduces a transformation group which is a submodule and a function apply than applies a transformation to a point. This function is used by tiles that depend on that geometry to implement their function visualize.

```
module type GEOMETRY =
sig
 type point
 type transformation
 module Tgroup : GROUP with type element = transformation
 val apply : transformation -> point -> point
end
```

For instance, to define hyperbolic geometries [3] (represented by its Poincaré model in Euclidean geometry), the transformations are complex homographies of the form

$$h_{a,\mu} = \mu \frac{z + a}{1 + \bar{a}z}$$

which we simply represent as a record with components μ and a.

```
module Hyperbolic_geometry  =
struct
type point = Mlgraph.point
type transformation = {iso_m:complex; iso_a:complex}
let identity = {iso_m=cx_1;iso_a=cx_0}
let compose {iso_m=mu1; iso_a=a1} {iso_m=mu2; iso_a=a2}=
  {iso_m= div_cx  (mult_cx mu1 (add_cx mu2 (mult_cx a1 (conjugate a2))))
              (add_cx cx_1 (mult_cx mu2 (mult_cx (conjugate a1) a2)));
   iso_a= div_cx (add_cx a1 (mult_cx mu2 a2))
              (add_cx mu2 (mult_cx a1 (conjugate a2)))}
 module Tgroup =
  struct
     type element = transformation
     let compose = compose
     let identity = identity
  end
let apply_cx {iso_m=mu; iso_a=a} z= ...
let apply iso pt = point_of_cx (apply_cx iso (cx_of_point pt))
......
end
```

The module contains other elements and in particular, definitions for specific transformations such as inversions and hyperbolic rotations, which are used to interpret the generators of formal hyperbolic groups.

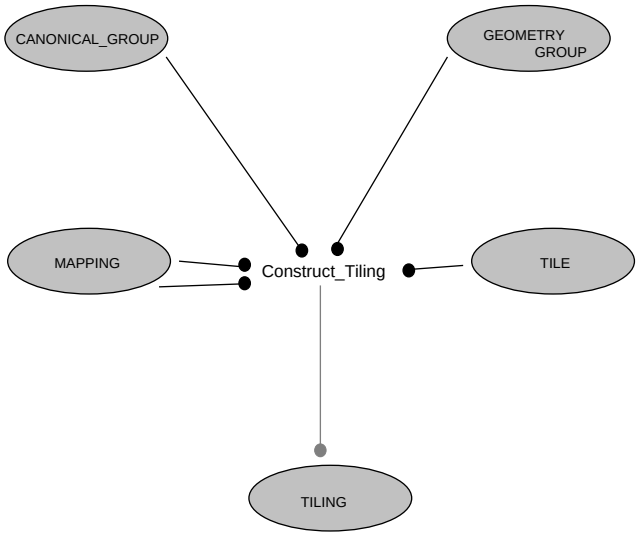

Fig. 3. Modules and functors.

Now we can summarize the construction of tilings in one functor that takes all the necessary ingredients as parameters:

```
module Construct_tiling
    (Group: CANONICAL_GROUP)
    (Geom: GEOMETRY)
    (GenMap: MAPPING with type source = Group.element
                     and type dest = Geom.transformation)
    (GenColorMap: MAPPING with type source = Group.element
                     and type dest =  Permutation.permutation)
    (Tile: TILE with type transformation =
                  Geom.transformation * Permutation.permutation)
=
Make_tiling
  (Make_generator_from_canonical
       (Make_canonical_generator(Group))
       (MakePair (Make_morphism (Group) (GenMap) (Geom.Tgroup))
              (Make_color_morphism (Group) (GenColormap)))
  (Tile)
```

This leads to a simplified graphical representation which appears in Fig. 3.

7. Conclusion

What we have obtained is a very generic program that can produce any tiling with a computable symmetry group that operate transitively on the tiles. For instance, the two

Fig. 4. Escher's Circle Limit III and a variant.

tilings in Fig. 4 are obtained by just switching the fish Tile module to a kangaroo Tile module (the symmetry groups are the same though the topological aspects are quite different). We can also obtain Euclidean tilings by changing the symmetry group into a Euclidean group, the geometry into Euclidean geometry and also changing the tile and the mappings.

This flexibility relies on the powerful programming constructs that have been used. Of course, this is only one specific example and we cannot infer from it how, more generally, mathematical structures should be implemented on computers. In particular, we had no need here for inheritance which would have been necessary if we had to deal with hierarchies of structures such as monoids, groups, rings, fields etc. But nevertheless, we think it shows at least that advanced programming constructs can help a lot in designing and implementing mathematical applications.

References

[1] E. Chailloux, P. Manoury, B. Pagano, Objective Caml, O'Reilly, 2000.
[2] G. Cousineau, M. Mauny, The Functional Approach to Programming, Cambridge University Press, Cambridge, 1997.
[3] H.S.M. Coxeter, Introduction to Geometry, Wiley, New York, 1980.
[4] H.S.M. Coxeter, W.O.J. Mauser, Generators and Relations for Discrete Groups, Springer, Berlin, 1972.
[5] The GAP Group, GAP: Groups, Algorithms and Programming, Version 4.2, 2000.
[6] B. Grünbaum, G.C. Shephard, Tilings and Patterns, Wiley, New York, 1987.
[7] M. Kofler, Maple: an Introduction and Reference, Addison-Wesley, Reading, MA, 1997.
[8] Ph. Lechenadec, Canonical Forms in Finitely Presented Algebras, Pitman, London, 1986.
[9] S. Wolfram, The Mathematica Book, 4th Edition, Cambridge University Press, Cambridge, 1999.